Horizon 2020
European Union funding
for Research & Innovation

SecureCloud

Joint EU-Brazil Research and Innovation Action
SECURE BIG DATA PROCESSING IN UNTRUSTED CLOUDS

https://www.securecloudproject.eu/

Description of dependability mechanism used by the micro-service framework
D3.3

Due date: 31 December 2018
Submission date: 31 December 2018

Start date of project: 1 January 2016

Document type: Deliverable
Work package: WP3

Editor: Stefan Köpsell (TUD)

Reviewer: Valerio Formicola (SYNC)
Keiko Veronica Ono Fonseca (UTFPR)
Luiz Celso Gomes Jr. (UTFPR)

Dissemination Level

PU	Public	√
CO	Confidential, only for members of the consortium (including the Commission Services)	
CI	Classified, as referred to in Commission Decision 2001/844/EC	

SecureCloud has received funding from the European Union's Horizon 2020 research and innovation programme and was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under grant agreement No 690111.

Contents

List of Figures

1 Introduction

Besides confidentiality and integrity, availability is another important protection goal the SecureCloud platform needs to address. Together the mentioned protection goals are the foundation of the dependability of the SecureCloud platform and the applications running on top of it.

How the SecureCloud platform achieves confidentiality and integrity was already discussed in several deliverables (e.g. D1.2, D3.2, D4.1, D4.3). Therefore, this deliverable concentrates on availability. Knowing that strong availability guarantees are impossible to achieve, if SecureCloud applications are executed in untrusted cloud environments (and therefore the cloud provider can simply withhold the necessary resources), we still designed many mechanisms which foster the availability of the applications. One fundamental building block is the microservice paradigm for developing SecureCloud applications since it inherently supports availability. Nevertheless, additional mechanisms are needed.

To introduce our dependability mechanisms, we first show the limitations of traditional dependability approaches like state machine replication and primary/backup approaches. Nevertheless, we still consider those mechanisms as a valid solution to ensure the availability of an **orchestrator** which in turn is used to ensure the availability of our SecureCloud services.

Despite not being able to trust the orchestrator regarding integrity and confidentiality, we trust the orchestrator regarding availability. The reason for that is that we can monitor the availability of services while it is practically impossible to monitor the integrity and confidentiality of unprotected applications. In this way, we can ensure availability by contracts between the operator of the orchestrator (typically a cloud provider) and the operator of the application (typically a service provider). The contract includes punitive fines that punishes the cloud provider in case it does not provide sufficient resources or does not replace failed resources fast enough to meet timeliness requirements of the user.

An orchestrator, despite being only responsible for availability, can in some cases violate the integrity of applications. For example, to enforce its invariants, many distributed applications depend on majority decisions, i.e., a majority of the application instances have to agree to perform a certain state change. For example, this is required to ensure consistency in the case of network partitions, i.e., that only the application instances in the majority partition can make progress. If an untrusted orchestrator could now spawn arbitrarily many application instances, even the instances in minority partitions could make progress and this would lead to data inconsistencies.

In the following sections we describe our approach that relies on the orchestrator to replace failed application instances but that verifies that the number of alive application instances is bounded by constraints given in the application's security policy.

In the context of the overall SecureCloud platform the presented dependability mechanisms are part of the SecureCloud runtime as well as the configuration service (a SecureCloud Infrastructure service) and interplay with the Scheduling and Orchestration services (most notable: Kubernetes). They can be utilised by all SecureCloud application microservices by extending the corresponding configuration files.

2 Background

2.1 Dependability

Dependability ensures the integrity, confidentiality and availability of services and applications. Trusted Execution Environments (TEEs) like Intel SGX help to ensure the integrity and confidentiality of applications but not their availability. To ensure their availability, we need to add additional mechanisms.

Traditionally, one uses approaches like state machine replication (SMR) [11] to ensure the availability of a service: the requests are executed by a set of replicas instead of a single instance. A service must be deterministic in the sense that the execution of each request results in the same result on all replicas as well as the same state changes in all non-crashed replicas. These days, most services are non-deterministic: in most cases, they use multi-threading and real-time clocks and operating system calls can return non-deterministic errors, e.g., caused by transient resource limitations. As a result, the states of the replicas in SMR may diverge, i.e., leading to false positives in the sense that the replicas produce different outputs despite all replicas being correctly executed.

To deal with non-deterministic behaviour, one could use a primary-backup approach [9] instead. A primary copy executes the requests and transfers its state updates to one or more backup replicas. In case the primary fails, one of the backup replicas takes over. A state corruption of the primary replica will propagate to all its backup processes. Note that in the state machine replication approach, disagreements can be detected and one can tolerate state corruptions as long as a majority of the replicas continues to agree.

A limitation of the state machine approach as well as the primary/backup approach is the limited fault model: they can tolerate crash failures of the underlying hosts and potentially transient miscomputations by CPUs (in some variants of the state machine replication) but they cannot tolerate software bugs. In case a software bug is triggered, a service is typically aborted and must be restarted from an initial state: both approaches ensure that the states of the replicas stay in sync and hence, software bugs will most likely be triggered in all replicas. Given modern development processes with continuous delivery of new software versions, one expects that triggering of software bugs is not unlikely.

2.2 Horizontal Scalability and Elasticity

A modern approach to dependability is to support multiple instances of a service. Each instance can process service requests independently of the other instances. The number of service requests that can be processed per second is therefore proportional to the number of service instances. Modern systems are elastic [8] in the sense that the number of instances can be adjusted to the current request rate: when the request rate increases, new service instances are spawned and if the request rate decreases, some service instances are stopped.

Supporting elasticity, requires the use of soft state [10], i.e., the instances keep state in main memory to improve the throughput but a crash of an instance will not result in a state loss. The persistent state is typically stored in a database or a key/value store. The databases and the key/value stores ensure the durability of the state.

2.3 Restartability

A crash failure of a service instance is addressed by a restart [7]. This works independently if the service has crashed because of a software bug or a crash of the underlying host. Note that in the case of a software bug, the service instance restarts from its initial state – which is typically, the best tested state. Hence, it is

unlikely that the same bug is triggered again in the new service instance since the state of the instance is reset on each restart and bugs of mature software is typically non-deterministic, i.e., difficult to reproduce after a restart.

3 Mechanisms

In the following sections we describe our approach to ensure that only a well defined number of instances of a given microservice runs at the same time or is ever started. The fundamental building block is a lease mechanism. But as such lease mechanisms depend on a reliable time we first describe how we can securely reason about the current time, even if we execute our microservices on untrusted operating systems and hypervisors.

3.1 Orchestrator

A modern approach to implement elasticity and restarts is to use a container orchestrator like Kubernetes [5]. The orchestrator detects when a service is down or unresponsive and restarts the service instances after failures. Moreover, such orchestrators can scale the number of service instances to adjust the current request rates.

The availability of the orchestrator itself might be achieved by using state machine replication or a primary/backup approach while services themselves must only be restartable, i.e., in case of a failure they can be restarted either on the same host or if that host has failed, on a different host. To reduce the number of components of the trusted computing base (TCB) [12], the orchestrator must not be part of the TCB – at least not with respect to confidentiality and integrity.

Therefore, our approach is to not trust the orchestrator with respect to integrity or confidentiality. We do, however, trust that the orchestrator performs sufficiently well to ensure availability. The reason for this difference in trust regarding these three protection goals can be explained as follows:

- it is difficult to detect as well as to handle violations of confidentiality. Leaking of data could have multiple sources and hence, it can be very difficult to pinpoint that the orchestrator (or, any other component) leaked the data.
- a similar situation is given regarding integrity. In complex services, it is often very difficult to verify the integrity of results and state changes. Monitoring the integrity of an application is difficult. It is even more difficult to detect that the cloud provider or the orchestrator caused an integrity violation. Moreover, it is difficult to deal with integrity violation since wrong results might have already been sent to clients. Also, it might be difficult to correct the state of a service once it is corrupted.
- monitoring the availability of service is, however, comparably easier to perform and to mitigate. Violations of resource allocation can be detected and the cloud provider can be contractually required to pay financial compensations.

3.2 Rely but Verify

As mentioned above, we rely on the orchestrator for ensuring availability, i.e., to start new service instances and to stop existing instances. However, some services define constraints regarding the minimum and maximum number of concurrent service instances and if a service instance is permitted to be restarted at all. Our general approach is to let a service provider specify the minimum as well as maximum number of service instances as well as defining if service instances are allowed to be restarted.

3.3 Singleton

For some services, there should exist at most one instance at a time. For example, we might only permit one instance to accept client requests and perform database updates. If this instance crashes or exits with an error code, it should be restarted. However, an adversary should not be able to start more than one instance at a time. We show below that this is actually difficult to enforce.

From a practical perspective, the SCONE Secret Service (Palaemon) (part of the SecureCloud infrastructure services) permits to specify for each service a property describing how many instances of that service could be started or run at the same time. This property is called instances in the configuration file. One execution mode is singleton which states that at most one instance at a time can execute in the context of a given session. Say, a client wants to render an image and specifies a session MyPicture that gives the rendering software – in this case, blender – access to some encrypted input file. To ensure that an adversary cannot run multiple blender instances at a time, we can specify that we only permit singleton execution:

```
name: MyPicture
digest: create

services:
   - name: blender
     image_name: sconecuratedimages/iexec:blender_python
     instances: singleton
     ...
```

3.3.1 Single Shot

For some services, we might want to have an even stricter execution mode than singleton: we might need to ensure that at most one instance is ever started. This instance cannot be restarted nor can any other instance be started. Even an adversary with root access cannot restart the service. This can, for example, be required in case of a service instance performing actions that should only be performed once like, for example, initializing the file system.

Another example is that we render an image or a movie. In this case, running the renderer a second time will not only incur costs for the cloud resources but might also expose the image to side channel and other attacks. An attacker might restart the application to attempt different sidechannel attacks on each rerun to get access to the rendered images / movies.

In the session file, we can specify the value of the property instances as single_shot. In this case, in the context of this session, the service can be executed at most once. In case an error has occurred during the initial execution of the service, the client could create a new session or change the value of the property instances to singleton.

```
name: MyImportantPicture
digest: create

services:
```

```
  - name: blender
    image_name: sconecuratedimages/iexec:blender_python
    instances: single_shot
    ...
```

3.3.2 *N* instances

In some cases, we want to bound the maximum number of running instances of a service. For example, we might run a distributed protocol amongst the instances that ensures that only the instances in a majority partition can make progress – which is for example the case in consensus protocols. In case a malicious orchestrator would be able to spawn more than N instances, instances in a minority partition could make progress independent of the majority of instances. This means that by spawning too many instances, an attacker might be able to violate the integrity of the service.

In the session policy, we can specify the maximum number of instances of a services by giving a natural, positive number as the value of the property instances:

```
name: MyConsensus
digest: create

services:
   - name: paxos
     image_name: sconecuratedimages/apps:paxos
     instances: 7
     ...
```

3.3.3 Implementation

As we already mentioned above, ensuring a maximum number of instances is not trivial. Let us start by showing how to ensure a single_shot instance.

Single Shot

Implementation of a single shot instance is straightforward. When any instance starts up, it must get attested by Palaemon such that Palaemon passes keys and other configuration information to the instance. Palaemon stores all session data in a database that is rollback protected. For single shot instances, Palaemon checks its database if this is the first instance. If this is true, it updates its database noting that the instance has been started and returns the configuration information to the instance.

Singleton

If we want to ensure that at most one instance executes at a time, it looks like we have to address an impossible problem: we have to determine if all previous instances of a service have stopped before we permit a new instance to get attested. In asynchronous distributed systems, this is actually impossible to determine. However, we can use a lease mechanism instead: during initial attestation, the instance gets a lease from Palaemon. As long as the instance can extend its lease, it will be permitted to continue to

run. If its lease expires, the instance will detect it (each instance has an internal trusted timer, see 3.4) and terminate. In the case of a singleton, Palaemon will give at most one lease at a time. In case of N instances, it will grant up to N leases, thus a maximum N instances can run at the same time.

One important technical problem that we need to address is that one cannot implement a lease mechanism with the trusted time provided by the Intel SGX SDK. We needed to implement an alternative.

3.4 Lease Mechanism

Leases are one of the most important mechanisms to build distributed algorithms and systems. In the context of distributed systems, a lease is an access permission for a resource issued for a limited period of time—a lease term [4]. They are heavily used in practical systems to ensure properties like having at most one leader or one instance of a service. We implement a trusted lease mechanism on top of Enclave-Interval Timer (introduced in Section 3.4.3). The mechanism consists of two types of nodes, granters and grantees. Both types run inside Intel SGX enclaves.

The issue is that the concept of time inside of an enclave is insufficient. We show how to implement a lease mechanism anyhow. This permits us to solve problems like ensuring that service instances are singletons.

3.4.1 Why Trusted Time is Insufficient

Intel SGX SDK provides the notion of trusted time, i.e., even an adversary with root access cannot manipulate this time base. In particular, this means that the clock has a clock drift rate which is bounded by some known constant ρ. The implementation uses the functionality of the ME (Intel management engine). In SCONE, we provide access to the Trusted Platform Module clock instead.

The trusted platform module (TPM) is a standard for cryptographic coprocessors providing security-critical functionality. The current version of the specification, TPM 2.0 [2], requires all TPM chips to contain a built-in trustworthy source of time (TPM clock). TPM supports remote attestation, where a TPM chip issues a digitally signed quote certifying, among others, the TPM clock value [3]. Hence, no adversary can modify the clock value undetectably.

The TPM clock is running only when the TPM is powered-on. Internally, the clock is a volatile value updated every millisecond and periodically synchronized with the non-volatile (NV) memory. The related non-volatile memory value is called NV clock. The TPM clock can only move forward [2]. However, if a power loss occurs before the synchronisation, the TPM will report the stale NV clock value. To detect the outdated read, the TPM clock contains a boolean flag indicating if the current clock value is guaranteed to be different from the previously reported one [1]. Additionally, the TPM clock contains information about the number of TPM restarts, TPM resumes, and TPM resets. Their change may indicate a discontinuity of the TPM clock. Finally, the TPM permits external software to adjust the TPM clock rate within ±15% [1]. This adjustment allows synchronising the TPM clock with the real-time, however, when exploited by an adversary, it may affect the accuracy of the time-stamping or violate lease contracts.

The TPM is, however, implemented off-chip and reading it has long latency: Our measurements show around 200ms round trip time. Thus, the clock reading rate is limited. Moreover, since all messages to and from the TPM pass through the operating system, an adversary with root rights can delay the clock reading arbitrarily. In particular, a skilled adversary can delay the execution of the enclave threads exactly

after a clock was read. This means when a thread inside an enclave processes the clock value, the only thing known is that this time is in the past. But we do not know how far it is in the past. In case of a lease mechanism, the clock value might indicate that the lease has not yet expired while in reality, the lease has already expired.

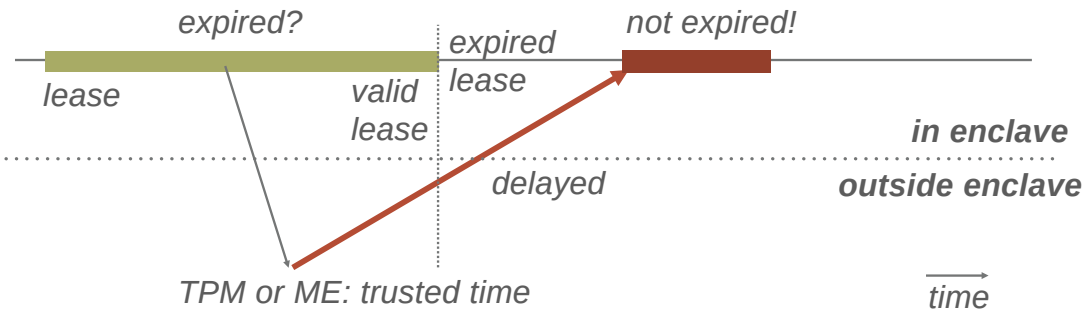

Figure 3.1: Example of an incorrect lease expiration check when using TPM or Intel ME. By delaying the return of clock reading, the speed of the clock is slowed down.

3.4.2 Problems

To implement a lease mechanism in enclaves, we had to address two technical problems:

- How can we increase the clock reading frequency from about 5 reads per second – when using the TPM clock – to a rate that can keep up with typical request rates, i.e., rates of a million requests and higher? In other words, we needed to improve the clock reading rate by 5–6 orders of magnitude while preserving the same security guarantees.
- How can we make sure that we can correctly decide if the lease has expired? An adversary can slow down the reading of a trusted clock (see Figure 3.1): In this way, the adversary can virtually slow down the speed of a clock and in turn, a client might wrongly decide that its expired lease is still valid.

3.4.3 Enclave-Interval Timer

Our core concept is Enclave-Interval Timer, a high-resolution low-latency timer that stays secure even in presence of an active privileged attacker. This high level of security, however, comes at the cost of lower precision in comparison to untrusted timers: If the measured interval is longer than the period between system interrupts, the timer provides only the safe estimation of the interval's lower bound (i.e., minimal estimation) and it determines ordering among measurements. But for shorter intervals, it has no restrictions.

As a basis, we use the Time Stamp Counter (TSC)[1], a standard timer embedded into the CPUs build on the x86 architecture. Although TSC has desirable properties of high resolution and low access latency, we cannot use it directly because TSC is under complete control of privileged software, either through

machine state registers (MSRs) or VM Control Structure. It makes the measurements unreliable from the security perspective.

We observe, however, that each core has its own TSC control registers. It means that the adversary cannot adjust the value or the frequency of TSC without either (1) preempting (interrupting) the process and taking over the core or (2) controlling a sibling hyperthread (see Figure 3.2). Hence, our approach is to rely on TSC when we are verifiably executing inside the enclave without potentially-malicious neighbors and revalidate the correctness of TSC measurements after every interrupt.

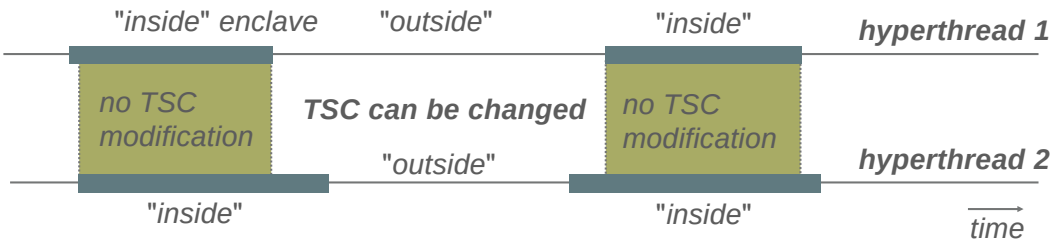

Figure 3.2: Safe Period

To implement this basic design, we have to solve the following problems:

- How to ensure that the adversary is not controlling the sibling hyperthread?
- How to safely estimate the duration of a time interval when the measurement is only partially trusted?
- How to detect interrupts?
- How to verify correctness of TSC after interrupts?

Safe Hyperthreading

There are two options for dealing with concurrent TSC modifications: ensuring that the adversary cannot control the hyperthread or disabling hyperthreading altogether.

To detect if not all hyperthreads of a core are allocated to our enclave, we could use the same approach as Varys [6]. However, this requires some compiler techniques to instrument the code running inside of the enclave, which is often incompatible with just in time (JIT) compilers.

Alternatively, we could instead enforce switching hyperthreading off (see Figure 3.3). We can enforce to have a single hyperthread per core with the help of the Intel Attestation Service (IAS). Since version 3, it can ensure that an attestation can only succeed if hyperthreading is switched off. This feature was introduced to mitigate side channel attacks using L1 and L2 caches. We make this feature available via the session policy. We can require that a service instance is only permitted to run on a core with hyperthreading switched off.

[1]Note that TSC is available to SGX enclaves only in the second iteration of the extension, SGXv2. Hence, we require this instruction set for our work.

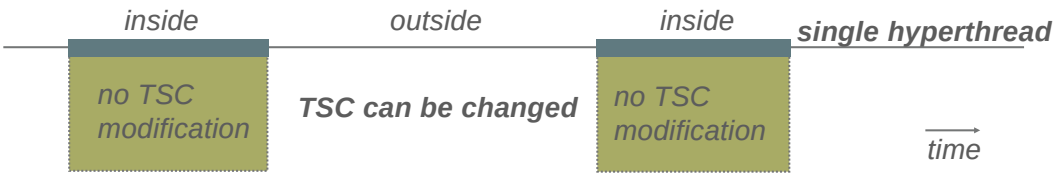

Figure 3.3: Single hyperthread per core

```
name: PreventHyperthreading
digest: create

services:
   - name: blender
     image_name: sconecuratedimages/iexec:blender_python
     instances: single_shot
     hyperthreading: off
     ...
```

By default, we switch hyperthreading off, when Palaemon needs to give an instance a lease. In other words, the value off of the property hyperthreading in the above example is already implied by the property single_shot.

Safely Estimating Time Intervals

To explain the idea behind safe estimation, we will use the example scenario presented in Figure 3.4. With Enclave-Interval Timer, the measurement precision depends on whether the interval fits into one epoch or it spans through multiple. Here, by epoch we understand an interval of time between two system interrupts. For example, requests 1 and 2 belong to different epochs, but request 2 and 3 are in the same epoch.

If the measured interval is in a single epoch (e.g., the interval $r2$), then the TSC readings can be trusted, as the timer is verified, and there are no interrupts to meddle with it. Yet, if the interval spans several epochs, we cannot reliably find out the duration of interrupts, and the TSC value could have been manipulated. Therefore, we have to estimate the duration as a sum of those periods that the application has spent inside the enclave. For example, we cannot reliably find out the interval between requests 1 and 2 as the length of $i1$ is unknown, but we can estimate it as a sum $r1\_1 + r1\_2$.

We implement this approach as shown on the listing below. The Enclave-Interval Timer periodically executes `rdtsc` instruction while checking for interrupts. In case there was no interrupt, we calculate the difference from the previous measurement, add it to the timer, and set the previous measurement as current. Otherwise, a recovery procedure is required.

```
int interrupt = <had_interrupt>;
current = rdtsc();
interrupt |= <had_interrupt>;
<check_results>
```

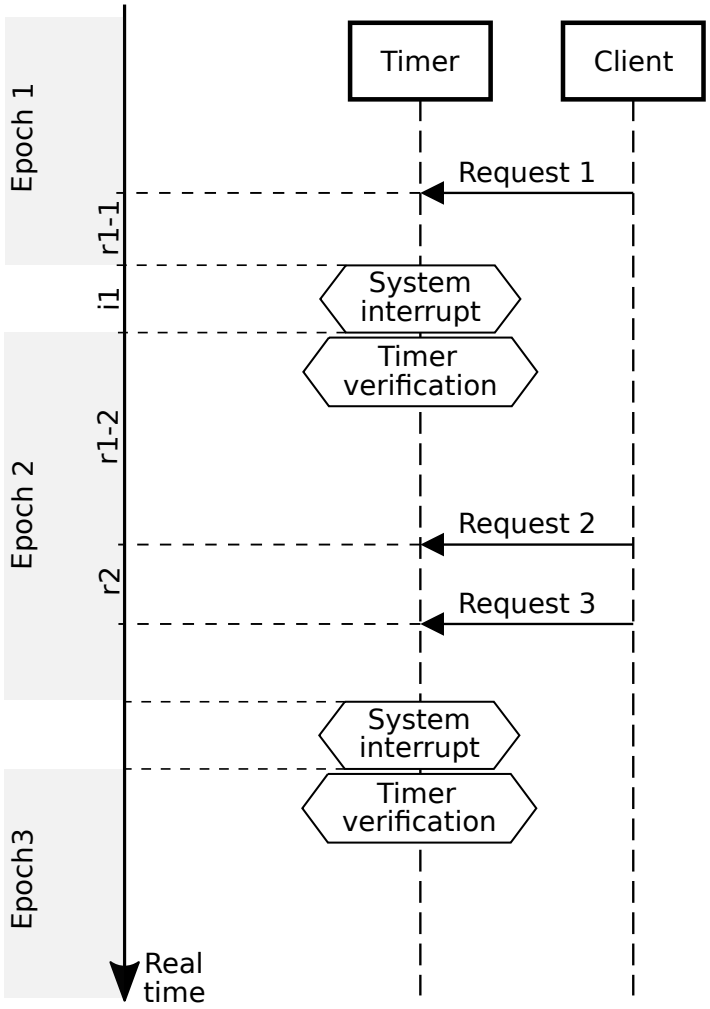

Figure 3.4: Example of interval estimation

```
<check_results> :=
  if (no interrupt) {
    time += current - previous;
    previous = current;
  } else {
    <recover from the interrupt>
  }
```

Detecting Interrupts

To detect an interrupt, we follow the mechanism proposed in Varys [6]. We use the fact that every interrupt triggers an Asynchronous Enclave Exit (AEX) during which the SGX hardware stores the current CPU state into the enclave memory (specifically, into the State Save Area (SSA)). We leverage this feature by writing a predefined value into the SSA and periodically monitoring it for changes.

First, we overwrite the SSA field that stores the Program Counter with zero (an invalid value of the counter). Then, when the interrupt happens, the CPU overwrites the SSA with the current register file state and the value in this field changes. Next time we read the field, we detect that it has changed and we interpret it as an indicator of an interrupt. We implement this simple mechanism as follows:

```
<had_interrupt> :=
  if (SSA.PC == 0) {
    return false;
  } else {
    SSA.PC = 0;
    return true;
  }
```

Timer Verification

Usually, when an interrupt is detected, the application has to perform a set of actions to recover the state that became stale or untrustworthy after the interrupt. Most of these actions are application-specific, but there is one generic procedure: ensuring that the clock drift rate was not modified by the attacker and we can still rely on it.

To verify the timer, we perform a reverse measurement by checking how many cycles it takes (accordingly to `rdtsc`) to execute a constant-time operation. We pick an operation whose execution time does not depend on the core frequency and, accordingly, the result of the measurement should stay the same given a fixed TSC rate. However, if the rate is changed, the result will differ thus indicating a potential attack. In this case, we abort the execution of the enclave.

```
<recover from interrupt> :=
  previous = current;
  <check timer frequency>
  <application-specific recovery>

<check timer frequency> :=
  start_value = rdtsc();
  <constant-time operations>
  end_value = rdtsc();
  measurement = end_value - start_value;
  if (measurement < lower_threshold ||
      measurement > upper_threshold) {
    error();
  }
```

3.4.4 Implementing Leases using Enclave-Interval Timer

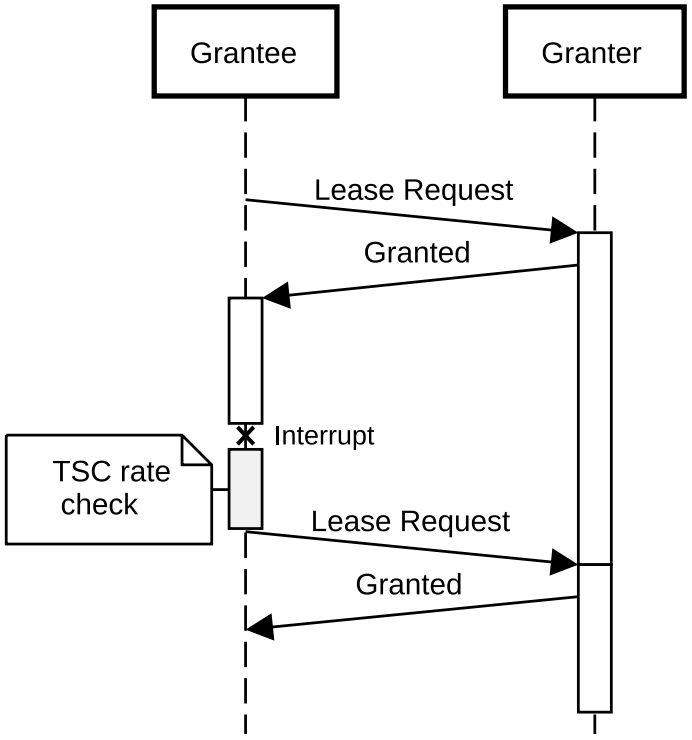

Figure 3.5: Trusted Lease Timeline

With the Enclave-Interval Timer, we can now implement the trusted lease mechanism.

Conceptually, we have two types of actors: a granter and grantees. The task of a granter is to maintain a set of leases for the grantees. Each lease is tracked using a global timer.

To acquire a lease, a grantee connects to the granter via a network socket and requests the lease with a specified ID (see Figure 3.5). It uses Enclave-Interval Timer to track the lease duration. When the lease is close to expiring, the grantee requests its extension. When the timer detects an interrupt, the grantee must recover. As we do not know how long the enclave has been descheduled, or how time has been manipulated, we cannot detect whether the lease has already expired. Therefore, after each interrupt, we request a lease extension to get the current lease status. Since we route interrupts to cores that do not execute enclaves, this approach introduces only very minimal overhead.

To estimate the message overhead of leases depending on the thread configuration, we measured the frequency of lease requests to the granter from the grantee (Figure 3.6). We can see that the frequency is determined by the minimum of lease duration and time interval between interrupts. With real-time threads, the interrupt frequency stabilises at 1.5 messages per second, whereas with nonrealtime threads, at 71.9 messages per second. The native case corresponds to the situation where we assume no interrupts happen, and the timer is trusted. It provides the lower bound on the achievable communication frequency.

In our evaluation, we run threads of the lease granter and grantee with normal and real-time threads. The reason for this choice is that threads running with the real-time scheduler (SCHED_FIFO scheduler

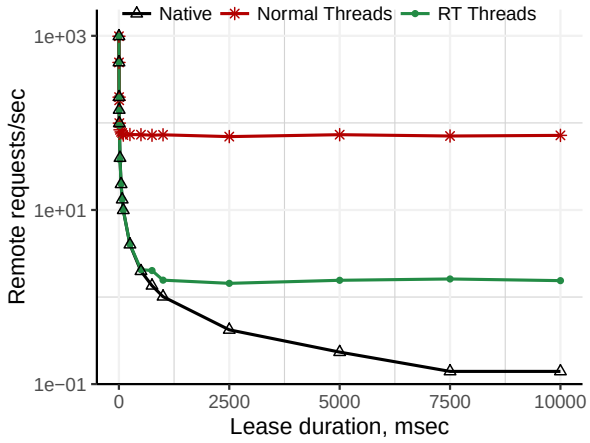

Figure 3.6: Frequency of remote requests.

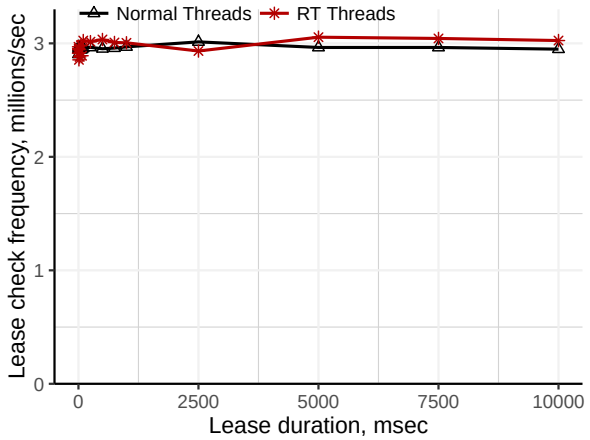

Figure 3.7: Lease check throughput at client.

in Linux) reduce the interrupt frequency. We measure the frequency with which lease grantee can check expiration of the lease. The results are presented on Figure 3.7. The frequency of checks has minor dependency on the lease duration, and no dependency on the scheduler used for enclave threads. There are two reasons for this:

- The fast path of the lease expiration check involves execution of `rdtsc` instruction and a few memory accesses.
- Remote communication is infrequent in cases of both real-time and non-real-time threads.

3.4.5 Device Access Control

In the context of cyberphysical systems, we might need to communicate via shared memory with other processes or might need to send commands to memory-mapped IO devices. To ensure that we only write

into that memory region while having a lease, we use hardware transactional memory. Hardware transactional memory is supported e.g. by Intel CPUs through the Transactional Synchronisation Extensions (TSX) instruction set extension. The commit of actions can be written to the shared memory in the context of a hardware transaction. By ensuring that the lease has not expired and will not expire for the duration of the hardware transaction, we can ensure that only the grantee can update the shared memory / device if and only if it has a lease. If the hyperthread is forced to exit the enclave during the hardware transaction, the transaction is aborted. This means that no shared memory state is changed. After the hyperthread reenters the enclave, the abort handler of the transaction is triggered which would check the validity of the lease before retrying the transaction. Note that it is easy to ensure that the hardware transaction is in the same epoch as the validity check. In other words, we can prevent that there is an enclave exit between the lease check and the commit of the hardware transaction (see Figure 3.8).

Figure 3.8: Lease Usage with Hardware Transactional Memory

With our lease mechanism, we try to prevent the attacks based on descheduling and changing time on either the granter or the grantee side, or changing the clock drift rate on either of the nodes. In case of the unprotected timer, this could lead to premature lease expiration, if the time on the grantee is counted faster than on the granter, and the impact would be increased load message traffic on the granter. If the time on the grantee is counted slower than on the granter, this could lead to the situation where lease expires on the granter before it expires on the grantee, which may cause a situation where several grantees hold the lease simultaneously.

4 Summary

The SecureCloud platform supports popular dependability mechanisms with the help of a standard orchestrator. The orchestrator takes care of the restarts while the secret management service SCONE Palaemon (previously known as SCONE CAS) of the SecureCloud platform, ensures that the orchestrator cannot spawn more instances than specified in the security policy of an application.

To be able to implement this approach, we introduced a new lease mechanism that has 5–6 orders of magnitude higher lease check rate than what would be possible with the standard trusted time provided by the Intel SGX SDK. Only through this lease mechanisms we can practically support leases and hence, dependability mechanisms: Intel's trusted time is too weak to achieve this.

Bibliography

[1] Trusted Platform Module Library Part 1: Architecture, Family "2.0", Level 00, Revision 01.38. http://www.trustedcomputinggroup.org/resources/tpm_library_specification, accessed on 15/06/2018.

[2] Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.38. http://www.trustedcomputinggroup.org/resources/tpm_library_specification, accessed on 28/11/2018.

[3] W. Arthur and D. Challener. A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security. Apress, 2015.

[4] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. SIGOPS Oper. Syst. Rev., 23(5):202–210, Nov. 1989.

[5] Kubernetes. Kubernetes. https://kubernetes.io/, 2018.

[6] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 227–240, 2018.

[7] Wikipedia contributors. Crash-only software — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Crash-only_software&oldid=858115792, 2018. [Online; accessed 17-December-2018].

[8] Wikipedia contributors. Elasticity (cloud computing) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Elasticity_(cloud_computing)&oldid=873266525, 2018. [Online; accessed 17-December-2018].

[9] Wikipedia contributors. Replication (computing) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Replication_(computing)&oldid=873308788, 2018. [Online; accessed 17-December-2018].

[10] Wikipedia contributors. Soft state — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Soft_state&oldid=861488231, 2018. [Online; accessed 17-December-2018].

[11] Wikipedia contributors. State machine replication — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=State_machine_replication&oldid=864039657, 2018. [Online; accessed 17-December-2018].

[12] Wikipedia contributors. Trusted computing base — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Trusted_computing_base&oldid=826284008, 2018. [Online; accessed 17-December-2018].